Currying & Higher Order Functions

Theory of Currying

* breaking down a function with multiple parameters into a series of functions that handle the parameters
* or the technique of converting a single function that takes multiple arguments into a series of functions that each take a single argument
* this also allows the creation of ***partially evaluated functions***
  + For example, instead of having a function add(2, 3), you would have a function *add(x)* that would be called twice: add(2)(3)
    - just calling add(2) would create a new function that always adds 2 to whatever argument is passed to it
* **not easily supported in most big languages**
  + **except JavaScript!!**

|  |
| --- |
| The visual way of viewing Currying |
| (this way normally)  (this way using Currying) |

|  |
| --- |
| JavaScript Currying Example |
| JavaScript Functions |
| function **add** (a, b) { return a + b; }  function **curry\_add** (a)  {  return function (b) { return a + b; }  }  function **curry\_factor** (a)  {  return function (b) { return a \* b; }  }  function **curry\_factor** (a) {  return function (b) {  // go through the array and factor  for(i = 0; i <b.length; i++)  { b[i] = b[i] \* a; }  // return the list of a \* b;  return b;  }  } |
| JavaScript “main” |
| print('Hello World');  print ('normal function call --> ' + add(3, 4) ); // returns 7  print ('curried function call --> ' + curry\_add(3)(4));  var add3 = curry\_add(3);  print ('broken up curried function call --> ' + add3(4));  //print ('curried function call --> ' + curry\_add(3)(4)(5));  // so try a curry\_factor example  var factor = curry\_factor(3);  print ('broken up curried function call --> ' + factor([4,6,8]));  print ('curried function call --> ' + curry\_factor(10)([4,6,8])); |
| JavaScript Output |
|  |

|  |
| --- |
| Other Language Examples |

|  |  |
| --- | --- |
| Scheme | Haskell |
| (define (curry f)  (lambda(x)  (lambda(y)  (f x y))))  (((curry +) 5) 5) | add :: (Num a) => a -> a -> a add first second = first + second  print ((add 5) 5) |

Convert the following Scheme function to a series of functions that only require one input.

(define (sub a b c)

(- (- a b) c)

)

(sub 23 13 3) +> (should look like) 🡺 ((sub 23) 13) 3)

What are higher order functions?

* Higher order functions can:
  + take functions as parameters
  + return produce functions
* are used for mapping, filtering, folding, and sorting of lists
* promote modularity of programs
  + Writing higher order functions that are applicable in many cases makes program readable rather than writing recursive functions for individual cases
* Higher-order functions are very common in Haskell/Scheme (and in functional programming).

|  |
| --- |
| Higher Order Function Examples |
| Gambit Scheme |
| (apply **char<?** lst) ; character list  (apply **string<?** lst) ; string list  (apply **<** lst) ; numeric list |
| C++ Version 1 |
| struct CompareNames{  bool operator() (const STUDENT &e1, const STUDENT &e2){  if(e1.Lname == e2.Lname){  return e1.Fname < e2.Fname;  }  return e1.Lname < e2.Lname;  }  }; |
| C++ Version 2 |
| friend bool compare(STUDENT &e1, STUDENT &e2) { return e1.score < e2.score; }  sort(CMSC\_202.begin(), CMSC\_202.end(), **compare**); |

Theory of Functions as parameters

* Design of the function is based on the application
* One object could be sorted (in our example) many ways

|  |  |
| --- | --- |
| **Function Design by Application** | |
| Version 1 (what could we compare?) | |
| Student 1 | Student 2 |
| Version 2 (what could we compare?) | |
| Student 1 | Lupoli |

In any applications we discussed, what return values would the sample function(s) return?

Why Are They Useful?

* Common programming idioms can be encoded as functions within the language itself
* Domain specific languages can be defined as collections of higher-order functions
  + For example, higher-order functions for processing lists.
* Algebraic properties of higher-order functions can be used to reason about programs

The ugly side of higher-order functions

* not always built-in
  + versions of Scheme vary widely!!
  + There is **no “filter”** in Gambit Scheme
* if you do not find the higher-order function you need, have to find it!
  + in GAMBIT – had to find them!
* once the code is downloaded, it then needs to be included with your code
  + in this case, we need to add to our Unix server!!

1. In our Unix server, create a directory “higherOrder”.
2. Download/Copy/etc… the files below into the new directory where you will be writing some Scheme code

wget <http://faculty.cse.tamu.edu/slupoli/notes/Scheme/code/apps.scm> .

wget <http://faculty.cse.tamu.edu/slupoli/notes/Scheme/code/sort.scm> .

|  |
| --- |
| Loading the custom higher-order functions |
| (load "sort.scm")  (load "apps.scm")  ;(display (map pow2 '(1 2 3 4))) ;pow2 is not in Gambit  (display (map + '(1 2 3) '(4 5 6))) |

1. Create a file higher.scm with the code above
2. Run the above code to make sure loading works
3. **What did the code do? What did it return?**
4. Add this line below:

(display (map + '(1 2 3) '(4 5 6 ***7***))) => (5, 7, 9)//!7

1. **What did it do? Any change in output? Why? (called type checking)**

The Map Function

* The higher-order library function called map applies a function to every element of a list

|  |
| --- |
| Scheme Map Function |
| Syntax |
| (map procedure list1 list2 ...)  ; notice this can be applied to multiple lists at a time |
| Simple |
| (map pow2 '(1 2 3 4)); gsi does not have this function (sadly)  ; Adding each item of '(1 2 3) and '(4 5 6).  (map + '(1 2 3) '(4 5 6))  ⇒ (5 7 9) |
| Complex |
| ; Squaring each item of '(1 2 3)  (map (**lambda** (x) (\* x x)) '(1 2 3)) ⇒ (1 4 9)  [1]  [0]  ; also had to do this for the basic Gambit-Scheme setup!! It didn’t have pow2!!  Notice how useful lambda is here, since it requires a function!  Also notice this is curried!!  [0] = lambda function (could have been anything!!) [1] = value |

1. Using the code already created and above, create the function “curve” that will accept a list (you create) and add 10% to the grades. Display your result for proof it worked.
2. Can you change your function slightly to use lambda?
   1. Did data change types? (welcome to type checking)

Answerb:

The Filter Function

* The higher-order library function filter selects every element from a list that satisfies a ***predicate***
  + ***the predicate function is a part of the requirements***

|  |
| --- |
| Scheme Filter Function |
| Syntax |
| ; **not in gambit scheme**, why we load sort.scm  (filter predicate-function list1) |
| Simple |
| (filter positive? '(1 -3 4 7 -4)) ⇒ (1 4 7) |

1. Using the Gambit Scheme HTML Manual, find 2 “check” functions (functions with ? at end). You might need to change the data to get it to work, but those are minor changes.
   1. Make sure to include “apps.scm” // apps has filter in it!

The Sort Function

* I think you know how this works!
  + ***but the LAST parameter is how you wanted it sorted by!***

|  |
| --- |
| Scheme Filter Function |
| Syntax |
| ; need to (include "sort.scm") for Gambit  (include "sort.scm")  (sort list1 comparison-operator/function) |
| Simple |
| (sort '(7 3 8 2) <) ⇒ (2 3 7 8) |
| Complex |
| 1. Complete a sort on a list of Characters! What comparison function should you use?    1. Make sure to include “sort.scm” 2. Now try Strings! |

The Apply Function

* applies a procedure to a list
* the first parameter
  + has to be the function
* last parameters
  + has to be the list
* everything in between is what is applied
  + this part is computed first!

|  |
| --- |
| Scheme Apply Function |
| Syntax |
| (apply **function** optional-values to be applied list1) |
| Simple |
| (apply **max** '(1 3 2)) ⇒ 3 |
| Complex |
| (apply **+** 1 2 '(3 4 5)) ⇒ 15  (apply **-** 100 '(5 12 17)) ⇒ 66 ; **100** – 5 - 12 – 17 = 66 |

Creating your own higher-order functions

* the name of the function can be anything you want
* but the first parameter should be a function that will be used to determine the result

|  |
| --- |
| Member-If Function |
| Description |
| returns a sub-list of items that fits the predicate function |
| Code |
| (define (member-if proc ls)  (cond  ((null? ls) #f)  ((proc (car ls)) ls)  (else (member-if proc (cdr ls))))) |
| Results |
| (member-if positive? '(0 -1 -2 3 5 -7)) ⇒ (3 5 -7) ; returns rest of list once it finds a  ; positive value |

1. Open <https://raw.githubusercontent.com/jlongster/gambit-iphone-example/master/lib/util/srfi-1.scm> (which is the apps.scm file, but full version).Using Gambit Scheme use 3 more higher functions:
   1. comment what they do
   2. use the higher order function found in the file
   3. display the result

Answers

|  |
| --- |
| Adding Ten Percent |
| (display(map (lambda (x) (\* x 1.1)) '(70 85 89))) |

Sources

<http://www.shido.info/lisp/scheme8_e.html>

Higher Order Function examples

<http://www.iro.umontreal.ca/~gambit/doc/gambit.html#Scheme-extensions>

Currying

<http://people.cs.aau.dk/~normark/prog3-03/html/notes/higher-order-fu_themes-curry-section.html>

https://stackoverflow.com/questions/36314/what-is-currying

Also thanks to Glen Hordemann (F’20)